Test Coverage Tools for Database Applications

Eric Tang
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11218
etang02@utopia.poly.edu

Phyllis G. Frankl
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11218
pfrankl@poly.edu

Yuetang Deng
Google, Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043
ydeng@google.com

ABSTRACT

Databases are everywhere. Whether it is commercial or scientific, a database is used. Therefore, it is essential that these database applications work correctly. Testing these database applications correctly and effectively is a challenge. AGENDA (A (test) GENerator for Database Applications) was designed to aid in testing database applications. In addition to the tools that AGENDA currently has, three additional tools were made to enhance testing and feedback. They are the log analyzer, attribute analyzer, and query coverage.The log analyzer finds relevant entries in log file produced by DBMS, lexically analyzes them using a grammar written in JavaCC, and stores some of the data in a database table. When the log entry represents an executed SQL statement, this statement is recorded. The attribute analyzer parses SQL statements. A SQL grammar for JavaCC was modified, adding code to determine which attributes are read and written in each SQL statement. A new test coverage criterion, query coverage, is defined. Query coverage checks whether queries that the tester thinks should be executed actually are executed. Similar to log analyzer, JavaCC was used to implement this. It is implemented by pattern matching executed queries against patterns representing abstract queries (including host variables) identified by the tester.

1. INTRODUCTION

Databases are very important tools that helps maintain information in an orderly fashion. Databases make storing and searching data quick and easy. No wonder more and more applications involve databases. Think of all the important information you retrieve from a database everyday. For example, checking your savings account balance, looking up your grades, searching for material on the internet, and much more. In many cases, these applications are mission critical. They are to perform correctly and efficiently because many critical operations rely on the data that is retrieve from a database. Because of the nature of these applications, testing the correct behavior of these applications is of great importance.

A relational database consists of a collection of tables, described by a schema. To free the programmer from worrying about low level description of data, the database management system (DBMS) translate high level data storage, access, and checking primitives to low level description of data. Application developers uses high level languages such as C, C++, Java, etc. to run queries to retrieve data for further processing. When the application executes the SQL statement, the statement is passed to the DBMS through API, executed by the DBMS, and the results are returned to the application.

The challenge in testing database applications is the possible states that the database can be in. Some challenging questions include: how to generate database states and applications inputs that throughly test relevant application behavior, did the test cases modified the database state correctly, how to asses test data adequacy, and how to describe test cases and their initial database state for archival and reuse purposes. The DBMS provides features that assists in tackling the chanlleges presented. In earlier work, we exploited the DBMS assertion checking mechanism to check whether tests modify database state as expected [6]. In this paper, we show how the DBMS logging mechanism can provide tremendous amount of information to determine test data adequacy. In addition to the challeges stated, applications with dynamic queries, such as many database applications written in Java, using the JDBC API to access the DBMS, pose additional challenges.

2. BACKGROUND

Traditional application testing techniques are not sufficient for testing database applications. Such techniques as statement, condition, and path coverage are primarily focused on imperative languages such as C/C++ and Java; they are inadequate for our purposes because they do not effectively test queries. Since queries, or statements that manipulate data in a database, are the most vital pieces of code in a database application, it is important that queries be tested effectively. Therefore, it is essential that we develop new techniques to test database applications.

Programs interact with the database through statements written in a language called Structured Query Language (SQL). SQL is used to create, modify, and retrieve data from a database. The queries on which we will focus in this paper are those that contain Data Manipulation Language (DML) commands. DML commands include SELECT, IN-

Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University, April 29, 2006

```
import java.sql.*;

class simple
{
        public static void main(String [] args) throws SQLException,ClassNotFoundException
{
 1.       Class.forName("org.postgresql.Driver");
 2.       Connection connect=
                    DriverManager.getConnection("jdbc:postgresql://localhost:5432/simple","eric","");

 3.       Statement simple = connect.createStatement();

 4.       int sup=Integer.parseInt(args[0]);
 5.       double prc=Double.parseDouble(args[1]);

 6.       String q="select * from coffees where SUP_ID="+sup;

 7.       if(prc>0)
 8.          q=q+" and price > "+prc;

 9.       simple.execute("COMMENT ON DATABASE simple IS \'_hot_spot_id 1\'");
10.       ResultSet rs=simple.executeQuery(q);
11.       while(rs.next())
{
                    // process results
}
}
```

Figure 1: Simple Example of a JDBC Application

test #	sup_id	prc	query
1	49	9.99	select * from coffees where SUP_ID=49 and price > 9.99
2	49	0	select * from coffees where SUP_ID=49
3	103	0	select * from coffees where SUP_ID=103

Figure 2: Test cases for simple example

SERT, UPDATE, and DELETE.

The commonly used application testing techniques, such as statement and branch coverage, do not measure which queries have been executed. This paper describes tools for measuring coverage according to criteria more directly based on the SQL statements in the program. Several such criteria have been developed recently [5]. We also introduce a new criterion that is targeted to programs with dynamically generated queries.

Testing database applications is a difficult task. To understand the problems involved, we must understand how a database application is written. In this paper, we focus on applications written in Java, using the JDBC library to connect to the database. In Java/JDBC, queries are treated as strings. The string (query) is passed to a function which passes that string into the database. The DBMS then processes the query. An error, if there is one, wont be detected during compilation; it will only be caught during runtime.

Throughout this paper we will illustrate with a simple example, based on the JDBC tutorial [3]. The database (for a coffee shop) has a table created by the SQL DDL statement

```
CREATE TABLE COFFEES (COF_NAME VARCHAR(32),
  SUP_ID INTEGER, PRICE FLOAT, SALES INTEGER,
  TOTAL INTEGER);
```

The table has columns for the coffee's name, the supplier ID, the price, the amount sold during the current week, and the total amount sold to date. The program shown in Fig 1 takes two arguments `sup_id` and `prc`, representing a supplier ID and a price. In lines 1 – 3, it sets up a connection and creates a `Statement` object. Lines 4 – 5 get the arguments from the command line. Lines 6 – 8 build up a string `q` representing a query, which is then executed in line 10. Line 9 is instrumentation explained in Section 3.1.1. Note that the form of the query string depends on how the `if` statement at line 7 evaluates. If the price is positive, the method returns the set of rows representing coffees that the supplier has with prices greater than the given price. Otherwise, it returns the supplier's coffees, without regard to price.

We assume that the application has a fault: assume the specification calls for returning coffees with prices less than `prc`*, i.e., that the "greater than" in line 8 should have been "less than".* In this simple example, any test case with `prc > 0` will detect the fault, provided that the `coffees` table has a row with the given `supplier_ID`.

Figure 2 shows three test cases for this example, along with the query executed.

The queries in Figure 2 can be thought of as strings that will be passed into the execute function. If there is an error in the query, the compiler will not detect it. The DBMS reports an error back to the database application after the DBMS executes the string that was passed into the execute function.

Section 3 describes three prototype tools to help the software tester determine how well the SQL statements in the program have been tested. These tools analyze the logs produced by the DBMS and compare the queries that were ac-

```
%%%%3937%%%%%LOG:  next transaction ID: 3186181; next OID: 2742086
%%%%3937%%%%%LOG:  database system is ready
%%eric%%3961%%43d56c1f.f79%%3186182%%LOG:  statement: COMMENT ON DATABASE simple IS '_hot_spot_id 1'
%%eric%%3961%%43d56c1f.f79%%3186183%%LOG:  statement: select * from coffees where SUP_ID=49 and price > 9.99
%%eric%%3961%%43d56c1f.f79%%0%%LOG:  unexpected EOF on client connection
%%%%3938%%%%%LOG:  checkpoints are occurring too frequently (29 seconds apart)
%%%%3938%%%%%HINT:  Consider increasing the configuration parameter "checkpoint_segments".
```

Figure 3: Excerpt from log file after executing Test Case 1

id	pid	sid	xid	qid	msg_type	msg	tc_id	hotspotid
1	3961			-1	0	COMMENT ON DATABASE simple IS '_hot_spot_id 1'	1	-1
2	3961			1	0	select * from coffees where SUP_ID=49 and price > 9.99	1	1
3	3961		0	-1	5	unexpected EOF on client connection	1	-1

Figure 4: Log Table entries from test case 1

.

tually executed to the queries that are potentially executed, as determined by analysis of the source code. In addition, potential database interactions, such as reads and writes, are compared with actual interactions. The potential interactions are currently determined by manual analysis of the source code at each execution point. [1] We refer to the program point where an execute function is a called as a *hotspot*. The source code is instrumented to record a unique id for each hotspot. This will be used later to trace back to which execute function was called to run the query. Soot is used to instrument a command that will indicate the hotspot id in the DBMS log.

2.1 JavaCC

To extract meaningful information from a query, we need a lexical analyzer and parser. JavaCC is a parser generator. In most languages, whether it is computer or human, there are grammatical rules that governs the proper use of the language. The language, in this case, is SQL. The grammar code for the SQL language is fed into JavaCC and creates the source code for the lexical analyzer and parser. A lexical analyzer breaks a continuous stream of characters into meaningful pieces called tokens. A parser is a program that analyzes the stream of tokens and compares it to a given grammar. Using the lexical analyzer and parser created by JavaCC, it is able to determine if a query is grammatically correct, if so, extract information about the query.

select	*	from	coffee	where	SUP_ID	=	49	;

The lexical analyzer separates the input query into blocks of text. The data in each box is called a token. The parser checks the tokens to ensure conformity to the grammar rules.

2.2 Soot

Soot is a Java Optimization Framework. It provides intermediate representation for analysis and transformation of Java bytecodes. The intermediate representation we are using is called Jimple. Soot converts Java class files (binary data called bytecodes) into a readable source code like representation of the bytecode. Modifications could be made to the class file without having the Java source code. Line 9 in Figure 1 is an example of instrumentation to indicate the hotspot's (execute function) id. The line 9 was included in the example to show what the program after instrumentation.

[1] In future work this will be partially automated.

3. COVERAGE ANALYSIS TOOLS

This section describes tools to analyze database logs, to compare the queries actually executed at each hotspot to those that are potentially executed, and to compare the interactions (reads and writes) that are actually executed with those potentially executed. We assume here that the relevant information from the source code has been previously determined (manually or through an automated approximation) and has been stored in DB tables.

3.1 Log Analyzer

The output of the log depends on the configuration of the DBMS. We have configured the DBMS to output more data then the default configuration.

The log table has entries for

- `id` unique ID for the entry
- `pid` process id,
- `sid` session id,
- `xid` transaction id,
- `qid` query id,
- `msg_type` message type (e.g., statement or error),
- `msg` raw message.
- `tc_id` test case ID
- `hotspotid` hotspot ID

Figure 3, is a sample of a log file after executing the program above in Figure 1 with testcase 1 in Figure 2.

3.1.1 Instrumentation

In producing the log files, the DBMS knows nothing about which hotspot a given SQL statement is executed from. Thus, we must instrument the program to cause hotspot IDs to be included in the log. During the log analysis phase, the executed SQL statement is associated with its hotspot.

auto_id	aid	qid	tid	tname	aname	context	condition	testcaseid
7	2	1	2796928	coffees	sup_id	read	where	1
8	3	1	2796928	coffees	price	read	where	1
9	1	1	2796928	coffees	cof_name	read	from	1
10	2	1	2796928	coffees	sup_id	read	from	1
11	3	1	2796928	coffees	price	read	from	1
12	4	1	2796928	coffees	sales	read	from	1
13	5	1	2796928	coffees	total	read	from	1

Figure 5: Executed Element table entries pertaining to test case 1.

.

In our current prototype, before each hotspot we add a special query involving the hotspot ID, as illustrated in line 9 of Figure 1. When testing our tools, line 9 was inserted manually. However, we have developed a tool that automatically inserts line 9 using Soot.

We are current working on a more elegant way to do this which is discussed later in Section 4. This new technique is not only more elegant but more flexible due to the ability to store more information.

3.1.2 Lexical Analysis

The first step is essentially a lexical analysis of the log file. For each relevant log entry, the process id, session id, transaction id, message type (e.g., statement or error), and raw message, are inserted into a `log_table` row. The raw message (i.e., the SQL statement executed or the error message) consists of multiple tokens. It's the last element of the log entry, so it can easily be copied in its entirety into the `log_table`. If the message is a SQL statement an ID is assigned so it can be referenced from the executed-element table, where more details about the statement's database interactions will be stored.

The hotspot ID is found by searching for the most recent log entry with the special hotspot query and with the same session, process, and transaction IDs. It is often the preceding log entry, but may be earlier in the log, due to statements executed by other sessions or by the test environment or additional messages from the DBMS.

Test case IDs are updated when the new test case markers are found in the log and are filled in at each `log_table` row. The `log_table` entries resulting from test case 1 are shown in Figure 4. Transaction and session IDs are elided to save space.

3.2 SQL statement analysis

Most of the coverage criteria discussed above involve analysis of SQL statements executed, to determine which database elements they involved and how those elements occurred. A SQL parser was modified to extract relevant information, map table and attribute names to unique IDs derived from the database's information schema, and store it in the database. We call the resulting tool component the *attribute analyzer*. The attribute analyzer fills an *executed element table* which has columns representing

- `aID` attribute ID from the information schema
- `qID` query ID from log analyzer
- `tID` table ID from the information schema
- `tname` table name
- `aname` attribute name
- `context` read or write
- `condition` where is the column name located (e.g. where, set, from, into, values)

Our attribute analyzer was implemented using JavaCC [1]. JavaCC inputs grammar rules and actions (written in Java) and produces a top-down parser, written in Java. We modified a SQL grammar from [2], adding actions to identify table and attribute names occurring in the SQL statement and to the context in which they occur (FROM clause, INTO clause, SET clause, WHERE clause, etc.) Methods corresponding to grammar non-terminals are modified to return lists of table and attribute names occurring in the token sequences they process.

The database schema is then used to find the unique ID for each relevant (table-name, attribute-name) pair. Attribute names can occur in SQL statments either with or without the corresponding table names. For example, one could have used dot notation to refer to `coffees.price` rather than `price`. Programmers usually write only the attribute name as long as it is unambiguous, i.e., as long as the same attribute name does not occur in more than one table involved in the statement. For each unqualified attribute name occurring in the statement, the attribute analyzer checks all the tables in the corresponding FROM clause, until a table with that attribute name is found.

For each attribute occurrence, a row is inserted in the executed-element table, indicating the attribute ID, the SQL statement ID, whether the attribute was defined (written) or used (read), and the clause in which it occurred. Although the coverage criteria described above do not distinguish between different ways an attribute may be used (e.g., in SELECT list, in WHERE clause, in ORDER BY clause, etc.), one can easily envision new criteria that use that information. Since we're collecting that information, anyway, we record it in the executed-element table, to allow flexibility for testers to define new coverage criteria. The part of the table resulting from analysis of test case 1 are shown in Figure 5.

3.3 Query Coverage

The tool Query Coverage compares queries actually executed at a given hotspot to the queries found in the source code at all execution points (hotspots). Query Coverage uses a parser to break queries into tokens. After each token, custom made Java code gets executed.

Before comparing, Query Coverage checks to see if the queries have the same number of tokens; if not, then the queries are not compared. If they are the same length, then

Query #	Query
1	select * from coffees where SUP_ID=? and price > ?
2	select * from coffees where SUP_ID=?

Figure 6: Possible queries that could of been executed

comparison of the tokens is made from start to end. In JDBC, the queries that are dynamic do not show values of the query run in the log. Instead, sybmols are used; therefore we must do pattern matching. The tool matches SQL keywords and operators. Any query that contains the pattern will be considered a match. The tool also produces a report that includes the number of queries extracted from the source code and the number of queries from the log. The queries that match, meaning they were run, are bolded. In addition, Query Coverage informs the examiner of how well the software is being tested, by reporting the percentage of queries covered.

Example

In Figure 1, there are two possible queries that can be executed as shown in Figure 6. The question mark indicates a value will be dynamically assigned during runtime. If test case 1 is used, query 1 will be executed. Likewise, if test case 2 is used, query 2 will be executed. Let's say test case 1 was executed, in the log, we will have query 1. The following tokens will be compared: Select, *, from, coffees, where, SUP_ID, =, and, price, >

4. FUTURE WORK

A better way to detect hotspots is to instrument a wrapper for the execute method. PreparedStatement is an interface, therefore, we can extend the interface and rewrite the functions in the class to store valuable data such as dynamically assigned values, testcase id, and etc during execution of the query. Every PreparedStatement and Statement will be replaced with our custom class called LogStatements.

For simplicity, functions setInt and setString will contain code that will extract the dynamically allocated value. All other functions in LogStatements model the same functionality as PreparedStatement. The Throwable class allows us to trace back to the line number of the execute function. This class will be implemented in the execute function of the LogStatement. The line number can also be utilized as the hotspot id. With this technique, it will no longer be necessary to instrument line 9 in Figure 1.

5. CONCLUSION

This paper addresses the question of how to construct tools to measure coverage. We point out that traditional approaches, in which tools essentially monitor control flow, are not suitable for database applications with dynamic queries. The new tools were designed to give valuable information to the software tester regarding code coverage. They use the results from static analysis, or analyzing code, and log analysis to inform the examiner how well the application has been tested.

The new prototypes will allow the tester to augment future test cases to produce improved coverage. This will result in better error detection, and enhance the performance and usability of the database application.

6. REFERENCES

[1] https://javacc.dev.java.net.

[2] http://www.cobase.cs.ucla.edu/pub/javacc/plsql/FormsPlSql.jj.

[3] *JDBC Tutorial*. http://java.sun.com/products/jdbc/book.html.

[4] http://www.sable.mcgill.ca/soot/. *Soot: a Java Optimization Framework*. 2002.

[5] G. M. Kapfhammer and M. L. Soffa. A family of test adequacy criteria for database-driven applications. In *ESEC/FSE*, Sept. 2003.

[6] Y. Deng, P. G. Frankl, and D. Chays. Testing database transactions with agenda. In *ICSE*, pages 78–87, 2005.